From Requirements to Implementations: A Seamless Development Process for Embedded Systems

A. Pnueli

The Weizmann Institute of Sciences
The John von Neumann Minerva Center for the Verification of Reactive Systems
and
New York University

Asian'99, December, 10, 1999

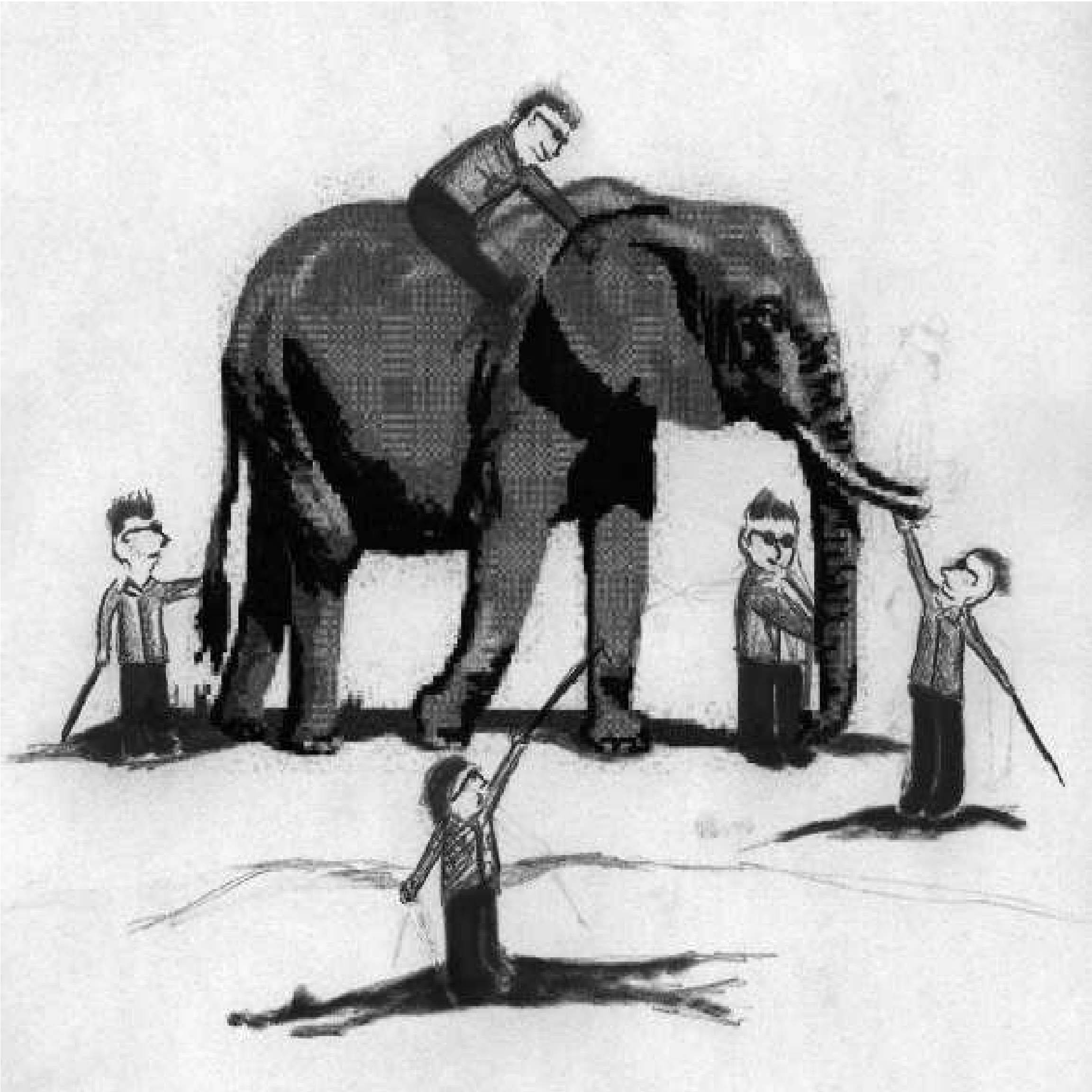
Embedded Systems

Everyone agrees that Embedded Systems will be one of the hottest items in the next millennium, both commercially and technologically.

However, this is where the consensus stops, and opinions differ as to what are the most important technological challenges whose conqueror will master the field. Sample items are:

- Communication and networking infrastructure and standards.
- Customizable and reconfigurable hardware.
- High-level programming & operating systems.
- Distribution and coordination.
- Effective and reliable development process.

What are the Challenges in Embedded Systems?



To What Extent are Embedded Systems an Emerging Application Domain?

Contrary to what some newcomers would have us believe, embedded systems are not a new technology. They have been with us for the last decades, embedded in the domains of telecommunication, aviation, automotive, and other process control applications.

If there is an embedded systems revolution, it is in the anticipated huge expansion of the embedded technology into (hitherto) dumb consumer products such as refrigerators, washing machines, etc. Questions are:

- Should we adopt one of the approaches currently endorsed by one of the existing embedded systems sectors?
- Is it time to introduce a horizontally integrating field of embedded systems with its own standards and techniques?

A Seamless Development Process: From Requirements to Implementation

An effective and reliable development process will support a fast turn-around and the ability to produce many customized variations of the same basic product.

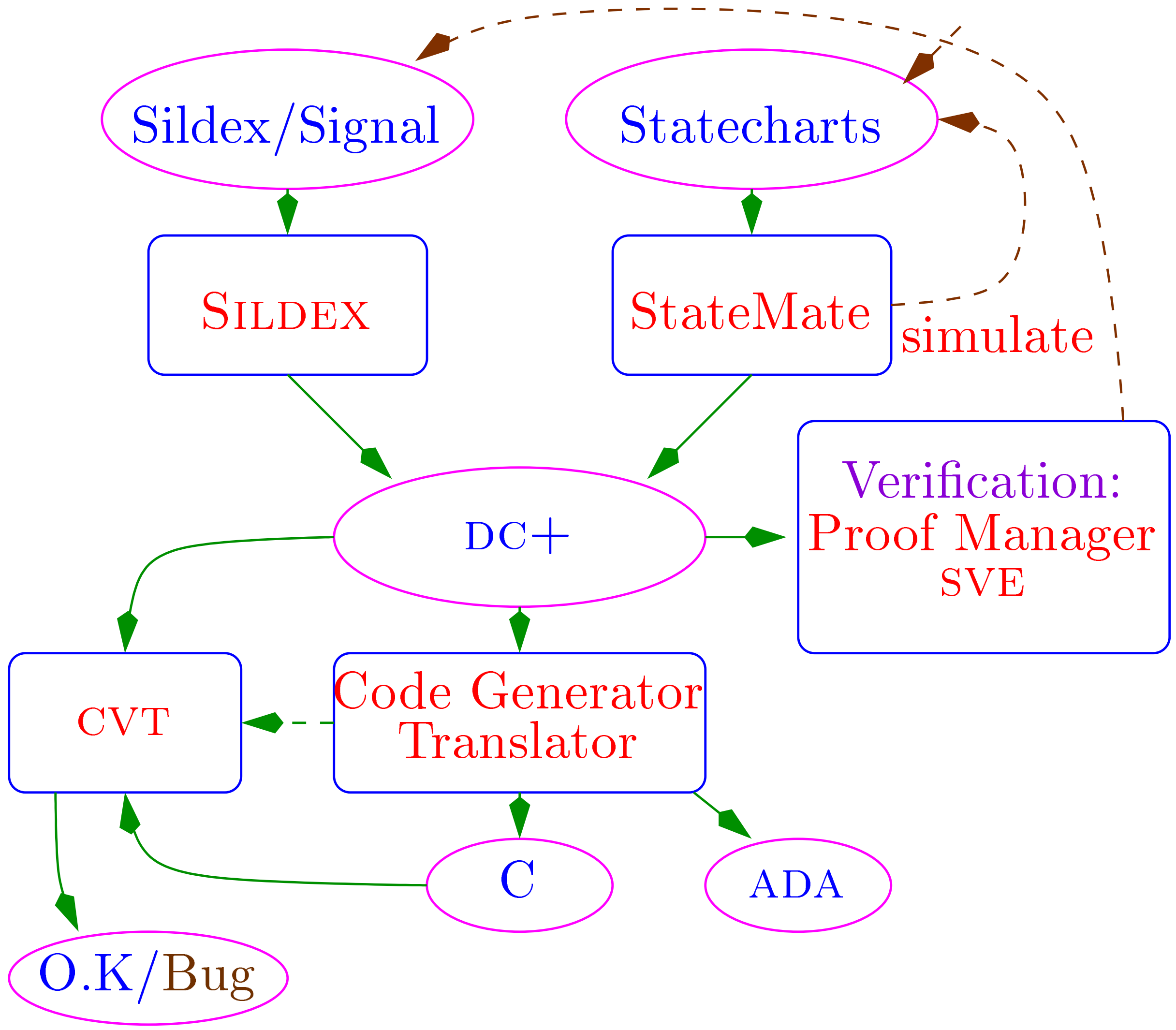
State of the Art: The *Sacres* Project

- Safety Critical Embedded Systems – From specification to architecture.
- A 4th framework European Community *Esprit* research project, 1995–1998.
- Participants

Industrial Partners
British Aerospace
Siemens
Snecma – A French jet-engine manufacturer
Tool Vendors
I-Logix U.K.
SNI – Siemens Nixdorf Information Systems
TNI
Technology Providers
Inria Rennes
OFFIS – U. of Oldenburg
Weizmann Institute of Sciences

The *Sacres* Mission

Developing a methodology and a tool suit supporting the development of safety critical real time embedded systems.

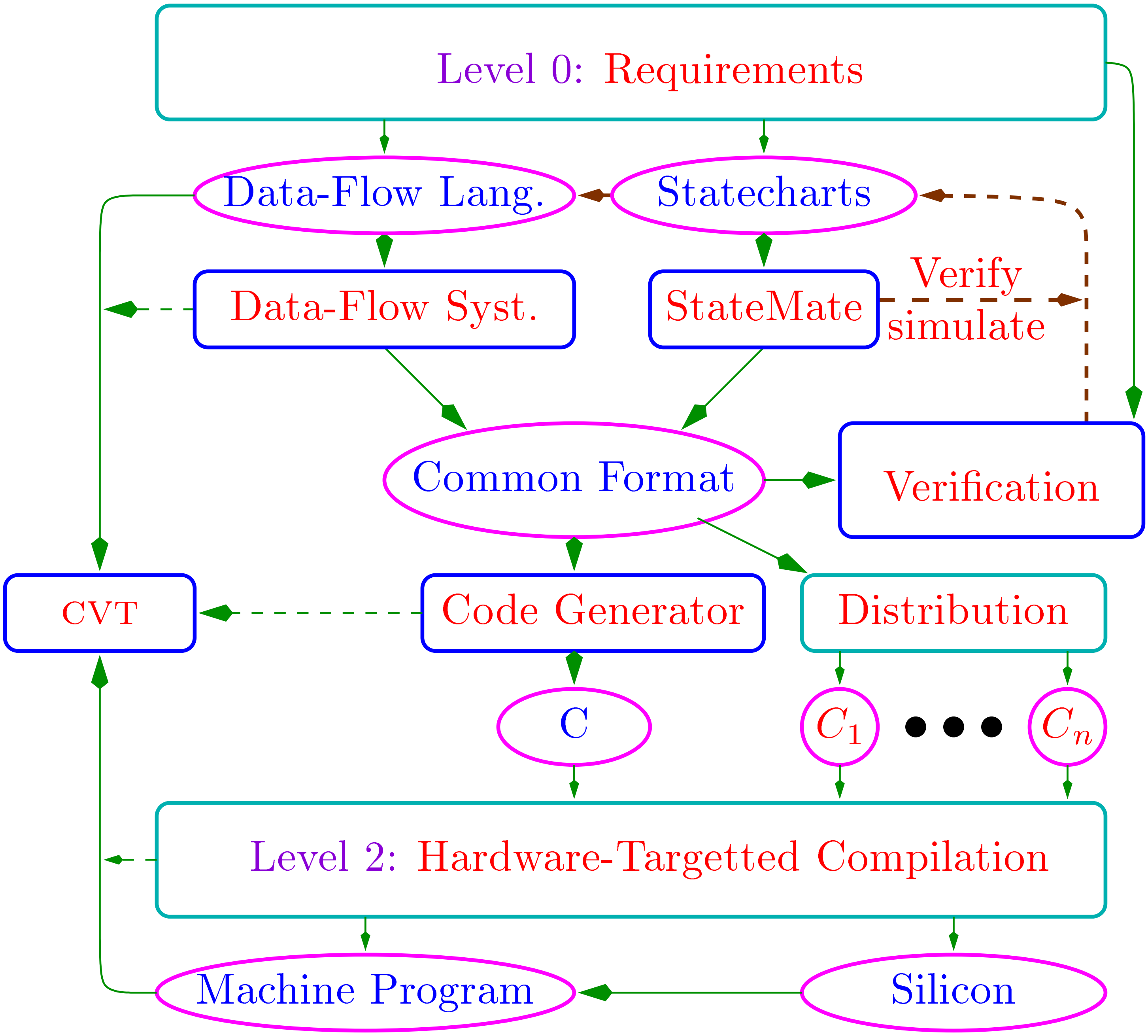


What is Missing?

The *Sacres* process covers only a middle layer in a more comprehensive and ambitious development process. The process can be greatly enhanced by adding:

- Level 0 – A requirement layer, in which user requirements are elucidated and crystallized into Statecharts/Sildex- level specification.
- Level 2 – A highly optimizing reconfigurable-architecture driven compiler (e.g. Trimaran) transforming C-level programs into machine-code/silicon.
- Code distribution decomposing a monolithic system description into a distributed implementation while synthesizing the necessary communication and coordination protocols between the components.

Achieve the Impossible Dream



Level 0: Requirements

Requirements, unlike specifications are not expected to provide a comprehensive integrated definition of the desired system behavior.

Rather, the process of requirement elucidation is an incremental process, possibly employing various different formalisms (languages). Every requirement adds a constraint which the system must satisfy. Possible languages:

- Temporal Logic or its timed variants, possibly in a style less threatening than

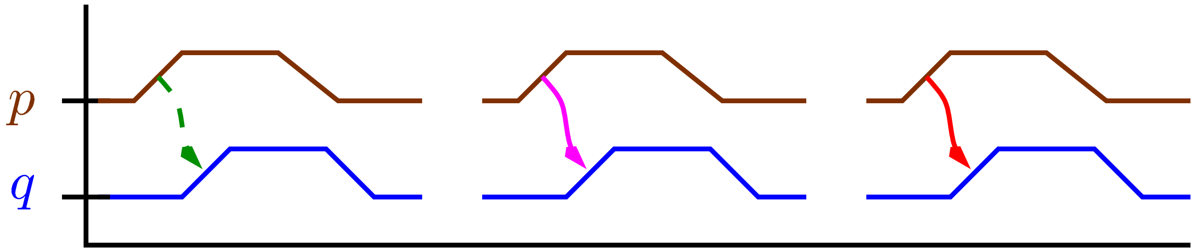
$$\Box \neg(p_1 \wedge p_2), \qquad \Box (p \rightarrow \Diamond q).$$

- Symbolic Timing Diagrams. These are wave forms with timing constraints and inter-wave arrows representing causality constraints.
- Live Sequence Charts – a more expressive variant [Harel and Damm 98] of the UML standard Message Sequence Charts.

Symbolic Timing Diagrams

This is a visual notation for expressing temporal requirements, originally developed by [Damm, Josko, Schlör, 95] and extended to include real-time constraints by [Feyerabend, Josko, 97].

The untimed version admits the following edges:



$p\uparrow$ must precede $q\uparrow$ $q\uparrow$ must follow $p\uparrow$ both

Can be translated into LTL as follows:

$p\uparrow$ must precede $q\uparrow$: $\overline{p} \wedge \overline{q} \rightarrow \overline{q}\,\mathcal{W}\,p$

$q\uparrow$ must follow $p\uparrow$: $\overline{p} \rightarrow \overline{p}\,\mathcal{W}\,(p \wedge \overline{q} \wedge \diamond q)$

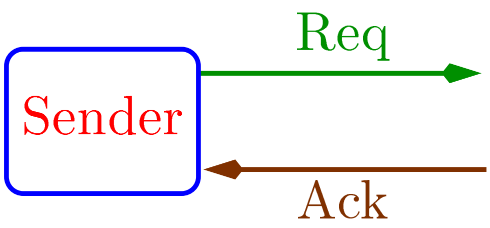
both:

$$\overline{p} \wedge \overline{q} \rightarrow (\overline{p} \wedge \overline{q})\,\mathcal{W}\,(p \wedge \overline{q} \wedge (p \wedge \overline{q})\,\mathcal{U}\,((p \wedge q) \vee \overline{p}))$$

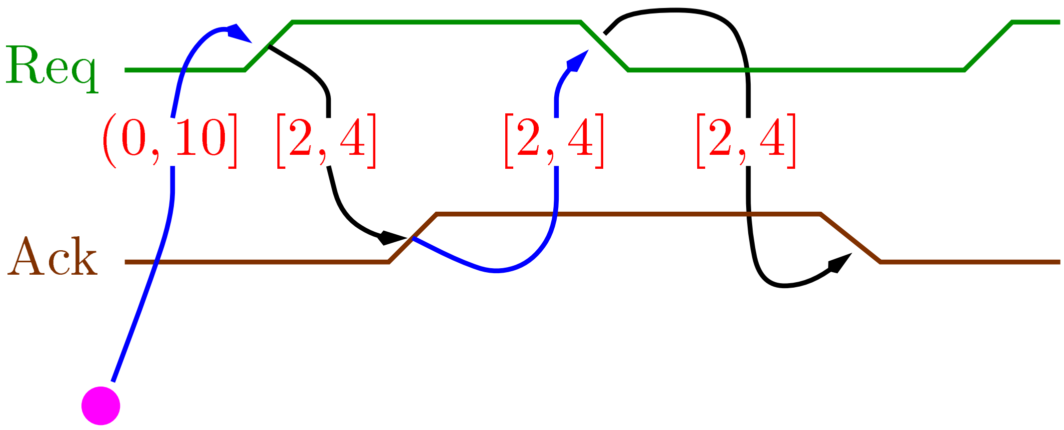
The Timed Version

We add interval timing constraints ($[low, high]$) to the edges, and distinguish between strong (guarantee) and weak (assume) edges.

We may specify the reactive module



by

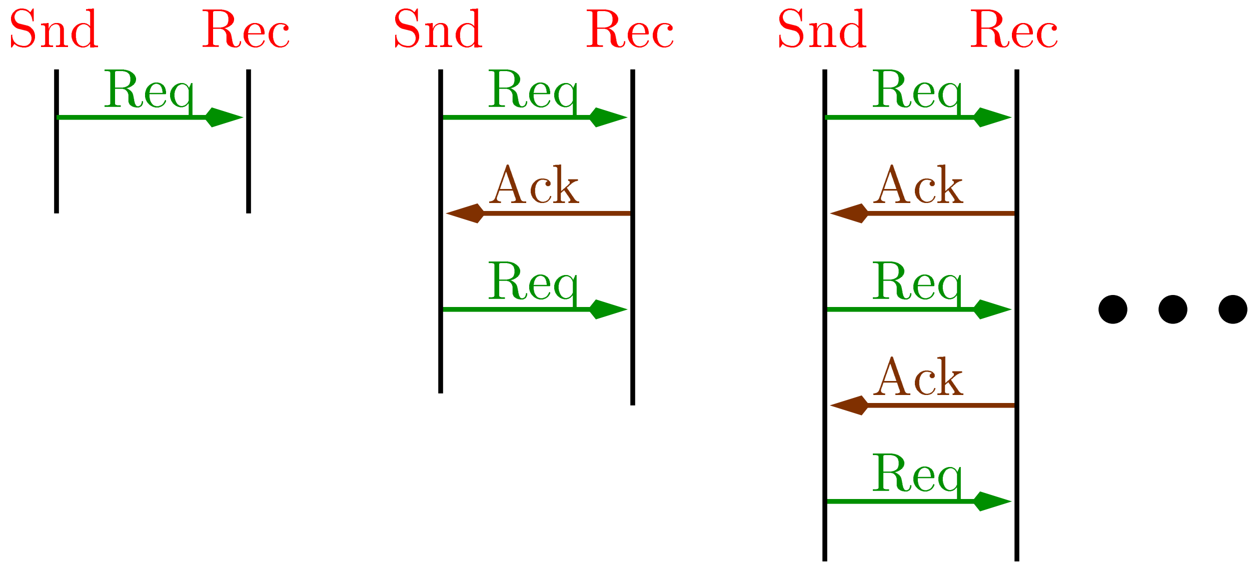


The first constraint can be temporally specified by

$$T = 0 \quad \to \quad (\overline{\text{Req}} \wedge \overline{\text{Ack}})\ \mathcal{U}\ (\text{Req} \ \wedge\ \overline{\text{Ack}} \ \wedge\ 0 < T \leq 10)$$

Message Sequence Charts (MSC)

A notation widely used for specifying use-cases in object-oriented methodologies, recently standardized as part of the UML standard. For example,

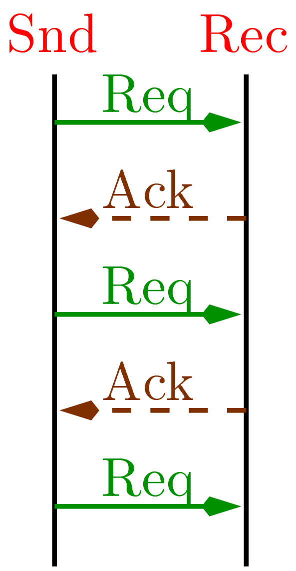


The problem with this specification is that it is completely existential. It only states that the system should at least have these scenarios. What about the missing ones? Are they forbidden or have they just been omitted?

This has the expressive power of CTL without the **A** path quantifier.

Live Sequences Charts (LSC)

In a recent work by Damm and Harel [99], they extend MSCs into LSCs, adding liveness and universality, attaining the expressive power of CTL*.

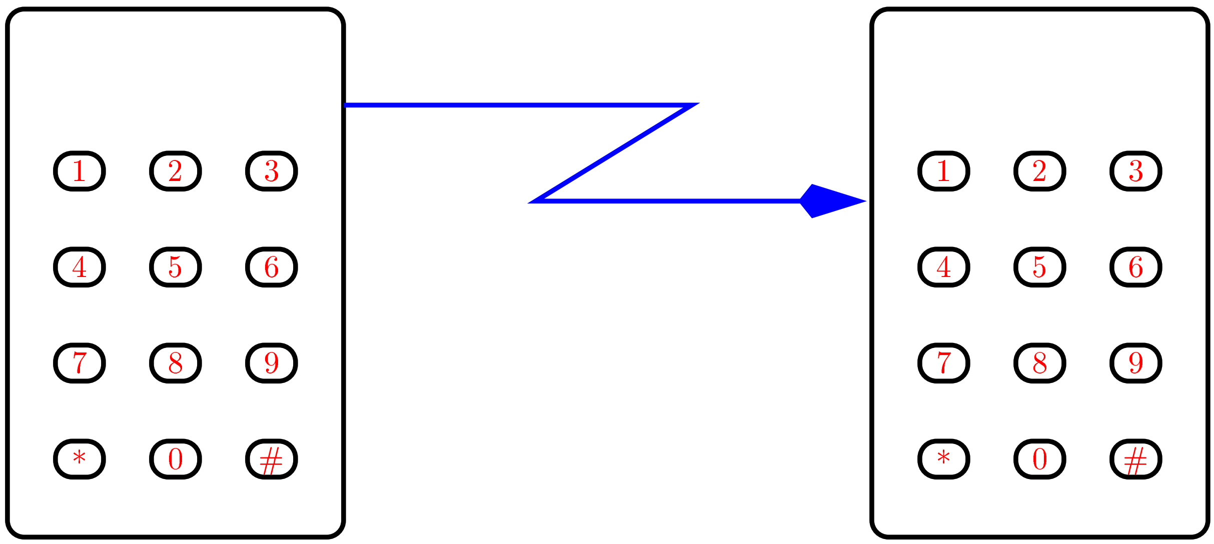


Identifying the Ack-edges as weak edges allows breaking the scenario at any such edge but nowhere else.

Fantasizing Even Further

The ultimate dream of requirement formulation may be based on a rapid prototyping system which will construct a virtual mockup of the driven system with the appropriate interface. The user will create scenarios (desired and undesired) by pushing the right buttons and manipulating the virtual model.

This will be gathered into am assembly of requirements, analyzed for consistency, serve as a base for user investigation, and finally translated into a specification.



Verification

A comprehensive approach to the formal verification of embedded systems can be based on a compositional verification methodology consisting of

- Unit verification performed by model checking technologies, plus
- System verification performed by deductive and theorem proving techniques.

Such a prototype system has been developed for the *Sacres* project by the OFFIS group, and is currently been adopted by Ilogix, the developer of Statemate.

An Example of Compositional Verification

Can be based on the idea of assume-guarantee. Consider program PING-PONG

$$
x,\ y,\ z : \textbf{boolean initially } x = y = z = 0
$$

$$
\left[\begin{array}{ll}
\multicolumn{2}{l}{\textbf{own x,z}} \\
& \\
\ell_0: & x := 1 \\
\ell_1: & \textbf{await y} \\
\ell_2: & z := 1 \\
\ell_3: &
\end{array}\right]
\quad \| \quad
\left[\begin{array}{ll}
\multicolumn{2}{l}{\textbf{own y}} \\
& \\
m_0: & \textbf{await x} \\
m_1: & y := 1 \\
m_2: &
\end{array}\right]
$$

$$
-\quad P_1 \quad - \qquad\qquad\qquad -\quad P_2 \quad -
$$

In order to verify the system property

$$
P_1 \parallel P_2 \models \Diamond \Box z,
$$

We perform unit verification of

$$
\begin{array}{l}
P_1 \models \Diamond \Box x \ \wedge\ (\Diamond \Box y \ \rightarrow\ \Diamond \Box z) \\
P_1 \models (\Diamond \Box x \ \rightarrow\ \Diamond \Box y),
\end{array}
$$

and then establish, using tautology checking,

$$
\Diamond \Box x \ \wedge\ (\Diamond \Box y \ \rightarrow\ \Diamond \Box z) \ \wedge\ (\Diamond \Box x \ \rightarrow\ \Diamond \Box y) \ \rightarrow\ \Diamond \Box z
$$

Code Distribution and HW/SW Allocation

A sound methodology for system development will initiate the development process viewing the system as a single unit. Identification of requirements, derivation of executable specifications, and a significant portion of verification can all be conducted at this level of abstraction.

Only when stability of the design at this level is attained, we can introduce more implementation details, such as partition of the design into its software and hardware components, and distributing the software into several sites, if the system is going to be implemented on a distributed platform.

Part of this allocation and code distribution can be supported by a certain degree of automation. For the *Sacres* project, the Inria-Rennes group developed a prototype tool which performs a code distribution process automatically, once the user has indicated the allocation of the main observable variables to the various sites.

This algorithm analyzes the dependency between the various variables residing in different sites, decides on the degree of synchronization implied by these dependencies and synthesize, wherever necessary, the communication protocols which perform the data transfer and synchronizes between the different units.

Extending the Scope of the Translation Validation Techniques

Naturally, when the development process is extended upwards by adding the requirement layer (layer 0) and downwards, adding the compilation into hardware layer (layer 2), it would be necessary to extend the scope of the translation validation techniques to cover these two far ends.

This raises many interesting questions and challenges to the presently used technique. We are currently experimenting with such extensions by constructing a translation validation tool for the Trimaran compiler.

Let us review the translation validation approach as it was implemented in the *Sacres* project.

Translator vs. Translation Validation

Rather than verify the translator itself (using compiler verification techniques), we chose to verify the results of each run of the translator.

Constructed the CVT tool, which can be invoked after every run of the code generator, translating a DC^+ specification D_1 into a C program C_1. Applied to (D_1, C_1), CVT verifies automatically that C_1 is a correct implementation of D_1 or identifies a bug in the translator.

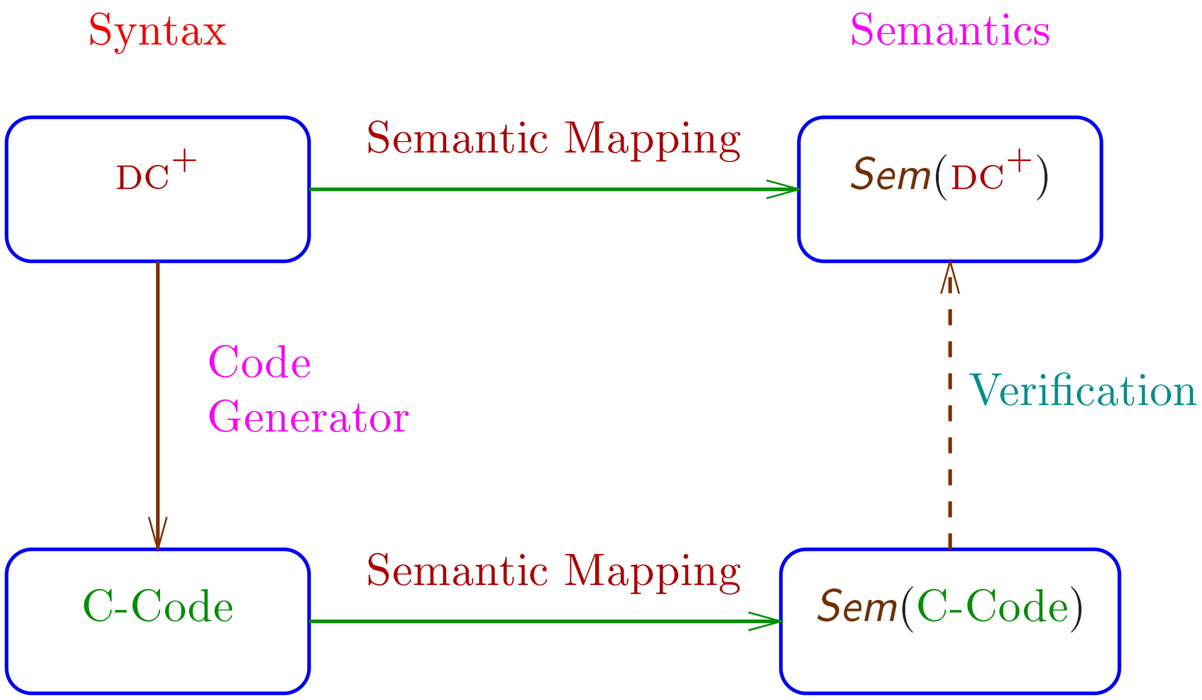
Advantages of the translation validation approach are:

- Evolution of the translator is not frozen after verification.
- Significantly cheaper than translator verification.
- Completely oblivious to the assumptions intrinsic to the translator. Provides independent cross check.

The only drawback: constant run-time additional cost, can also be justified.

The Framework for Formal Compiler Validation

The following diagram can be completed:



Formally Prove:

$$Sem(C(D)) \sqsubseteq Sem(D)$$

Every concrete step can be matched by an abstract step.

The Conditions that Made This Feasible: 1. The Source Language is Synchronous

This makes the application proceed in well-defined steps, as follows:

$$\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} \textbf{read inputs} \\ \textbf{compute} \\ \textbf{write outputs} \end{array} \right] \end{array} \qquad \begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} \textbf{compute; read inputs} \\ \quad \cdots \\ \textbf{compute; read inputs} \\ \textbf{compute} \\ \textbf{write outputs} \end{array} \right] \end{array}$$

A Statemate Behavior — A Signal/Sildex Behavior

Consequently, the generated C program has also only a single external loop. To verify correct implementation it suffices to prove an implication

$$\rho_C \quad \rightarrow \quad \rho_S,$$

where ρ_S is the transition relation of a single source step, while ρ_C is a (compressed) transition relation representing the state change effected by a single execution of the C-program loop's body.

The Conditions that Make This Feasible: 2. No Arithmetical Optimizations are Performed by the Compiler

For example, the assignment $y := 2x$ is never compiled into $y := x + x$.

This enables us to automatically verify the verification condition $\rho_C \rightarrow \rho_S$ without having to reason about arithmetical operations, using the decidable theory of uninterpreted functions.

The Principle Underlying Proofs of Refinement, using Uninterpreted Functions:

Every concrete application of a function f to arguments $t_1, \ldots, t_n$ corresponds to an abstract application of the same function to equal arguments.

As a running example, assume the the compiler translated $z := (x_1 + y_1) * (x_2 + y_2)$ into
$u_1 := x_1 + y_1;\ u_2 := x_2 + y_2;\ z := u_1 * u_2$. The corresponding verification condition is

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 u_2 \quad \rightarrow \quad z = (x_1 + y_1)(x_2 + y_2)$$

The Decision Procedure: 1. Abstract into Uninterpreted Functions

Starting with the implication $\rho_C \rightarrow \rho_S$, whose validity we wish to verify,

1. Abstract into uninterpreted functions: replace each concrete arithmetical operation by a fresh function symbol.

Applied to

$$u_1 = x_1 + y_1 \wedge u_2 = x_2 + y_2 \wedge z = u_1 u_2 \quad \rightarrow \quad z = (x_1 + y_1)(x_2 + y_2),$$

where $+$ and $*$ are replaced by F and G, this yields

$$u_1 = F(x_1, y_1) \wedge u_2 = F(x_2, y_2) \wedge z = G(u_1, u_2) \quad \rightarrow \quad z = G(F(x_1, y_1), F(x_2, y_2))$$

Note that this transformation strengthens the statement to be proved. We validate the translation not only for the particular operations $+$ and $*$ but for all functions F and G.

Subsequent Steps

The last reduction leads to a big implication in the quantifier-free fragment of the uninterpreted predicate calculus. Such formulas are decidable, and we have developed a very efficient procedures for their verification.

Experimental Results

A design of SNECMA turbine engine with *Sildex* results in a verification condition of about 6000 lines. Following is a summary of the results achieved by CVT:

Module	Conjuncts	Time (min.)
M1	530	1:54
M2	533	1:30
M3	124	0:27
M4	308	2:22
M5	860	5:55
Total :	2355	12:08

Extensions to More General Source Languages and Aggressively Optimizing Compilers

When we move to more general source languages, we lose the first simplifying assumption:

> The source and target programs consist of a single external loop, with $1-1$ correspondence between iterations of that loop.

To deal with program having nested loops we have to identify break-points b_i, such that every infinite computation can be broken into a concatenation of paths, each leading from some b_i to some b_j and not visiting any other break-point on the way. The main conclusion is that we need some information from the compiler, including

- A symbol table proving the mapping between source and target variables.
- A translation table establishing $1-1$ correspondence between the source break-points and their corresponding target locations.
- Established invariants at the head of basic blocks, capturing the results of data-flow analysis, and supporting all the optimization steps, such as code motion and constant propagation.

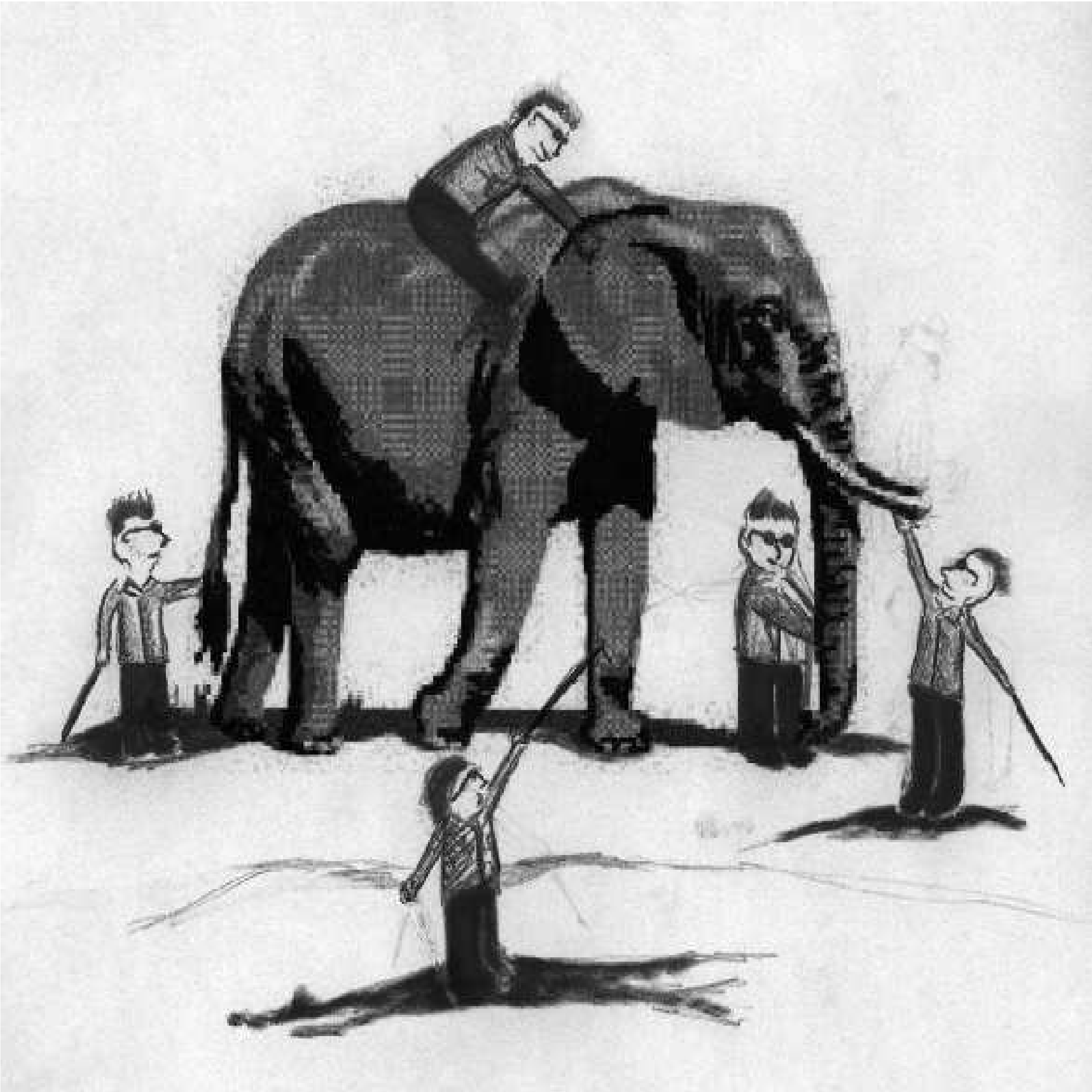
In Summary

- Without any doubt, we are entering a new era in which embedded systems will occupy a central and prominent place. There will be a great pressure to improve our capabilities for the quick and reliable construction and deployment of embedded systems.

- There are many currently existing techniques which propose partial solutions to the main problems intrinsic to embedded systems.

- To promote the progress of the field, it is necessary to enhance and strengthen the existing techniques, possibly develop additional ones and, most importantly, integrate all of these tools into a seamless development methodology.

Reinterpret the Elephant Picture



Experts, each in their own field, cooperate to achieve a better integrated methodology for the development of embedded systems.